www.arcomem.eu

ARCOMEM: ARchive COmunities MEMories

Collaborative Project (ICT-2009-270239)

Priority: FP7-ICT-2009-6 Challenge 4 – "Digital Libraries and Content"

D7.1ARCOMEM Development Guideline

Deliverable Co-ordinator: **Dimitrios Tsoumakos**

Deliverable Co-ordinating Institution: **ATHENA**

Other Authors: **Yannis Stavrakas, Radu Pop, Gideon Zenz**

Document Identifier:	ARCOMEM/2011/D7.1/v1.0	Date due:	31.3.2011
Class Deliverable:	ARCOMEM EU-ICT-2009-270239	Submission date:	31.3.2011
Project start date:	January 1, 2011	Version:	V1.0
Project duration:	3 years	State:	Final
		Distribution:	Public

2010–2011 © Copyright lies with the respective authors and their institutions.

ARCOMEM Consortium

This document is a part of the ARCOMEM research project funded by the ICT Programme of the Commission of the European Communities by the grant number ICT-2009-270239. The following partners are involved in the project:

The University of Sheffield (USFD) – Coordinator Department of Computer Science Regent Court 211 Portobello Sheffield, S1 4DP United Kingdom Contact person: Hamish Cunningham, Wim Peters E-mail address: arcomem-coord@lists.dcs.shef.ac.uk	**Leibniz Universität Hannover (LUH)** Forschungszentrum L3S Appelstrasse 9a 30169 Hannover Germany Contact person: Thomas Risse E-mail address: risse@L3S.de
Yahoo Iberia SLU (YIS) Avinguda Diagonal 177, 8th floor, Barcelona, 08018, CAT, Spain Contact person: Alejandro Jaimes E-mail address: ajaimes@yahoo-inc.com	**Internet Memory Foundation (EA)** 45 ter rue de la revolution 93100 Montreuil France Contact person: Julien Masanes E-mail address: julien@internetmemory.org
University of Southampton (SOTON) Room 4011, Building 32 Highfield campus University of Southampton SO17 1BJ Contact person: Paul Lewis E-mail address: phl@ecs.soton.ac.uk	**Athens Technology Center(ATC)** 10, Rizariou Street 15233, Halandri Athens, Greece Contact person: Dimitris Spiliotopoulos E-mail address: d.spiliotopoulos@atc.gr
ATHENA Research and Innovation Center in Information Communication & Knowledge Technologies (ATHENA) National Technical University of Athens School of Electrical and Computer Engineering Division of Computer Science Iroon Polytechniou 9 Athens, 15780 Greece Contact person: Nectarios Koziris E-mail address: nkoziris@imis.athena-innovation.gr	**Telecom ParisTech (IT)** 46 rue Barrault 75634 Paris Cedex 13 France Contact person: Pierre Senellart E-mail address: pierre.senellart@telecom-paristech.fr
Deutsch Welle (DW) Neue Medien / Distribution Voltastr. 6 13355 Berlin, Germany Contact person: Birgit Gray E-mail address: birgit.gray@dw-world.de	**SUDWESTRUNDFUNK (SWR)** Hans-Bredow-Strasse, D-76522 Baden-Baden Germany Contact person: Robert Fischer E-mail address: robert.fischer@swr.de
HELLENIC PARLIAMENT (HEP) Amalias 14, 10557, Athens Greece Contact person: Dimitris Koryzis E-mail address: dkoryzis@parliament.gr	**PARLAMENTSDIREKTION(AUP)** Dr. Karl Renner-Ring 3 A-1017 Vienna Contact person:Guenther Schefbeck E-mail address: guenther.schefbeck@parlament.gv.at

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed to the writing of this document or its parts:

ATHENA Research and Innovation Center in Information Communication & Knowledge Technologies (ATHENA)

Internet Memory Foundation (EA)

Leibniz Universität Hannover (LUH)

Change Log

Version	Date	Amended by	Changes
0.1	18-02-2011	Dimitrios Tsoumakos, Yannis Stavrakas	Initial Structure
0.2	22-02-2011	Dimitrios Tsoumakos	ATHENA contribution
0.3	25-02-2011	Gideon Zenz	LUH contribution
0.4	21-03-2011	Radu Pop, Dimitrios Tsoumakos, Yannis Stavrakas	IMF contribution, integration and completing sections
1.0	28-03-2011	Bogdan Cautis, Dimitrios Tsoumakos, Yannis Stavrakas	Internal review and final corrections

Executive Summary

To enable the multiple international ARCOMEM software development teams to work together effectively, a software development infrastructure has to be setup at the start of the project. This deliverable comprises an introduction of the development infrastructure, describing the tools that will be used during the design, implementation and deployment phases of the project. It also contains guidelines and recommendations on how to collaborate within ARCOMEM and use the tools involved in the software development lifecycle.

2010–2011 © Copyright lies with the respective authors and their institutions.

Table of Contents

1 INTRODUCTION 6
1.1 Scope 6
1.2 ARCOMEM Project Requirements 6
1.3 Document Structure 7
2 ARCOMEM SOFTWARE DEVELOPMENT INFRASTRUCTURE 8
2.1 Version Control System: Mercurial 8
2.2 Build and Integration Platform: Hudson 9
2.3 Bug Tracking System: JIRA 9
3 ARCOMEM SOFTWARE DEVELOPMENT BEST PRACTICE GUIDE 10
3.1 Version Control Management 10
3.2 Automated Building and Testing 10
3.3 Features/Bug Tracking 11
3.4 Coding Guidelines 12
3.5 Documentation 12
3.5.1 Product (Software) Documentation 12
3.5.2 Documentation Tools 15
4 CONCLUSIONS 16
5 REFERENCES 16
ANNEX A. JAVA PROGRAMMING CODING STANDARDS 17
A.1. FILE NAMES 17
A.2. FILE ORGANIZATION 17
A.3. JAVA SOURCE FILES 17
A.4. LINES 19
A.5. COMMENTS 21

A.6. DECLARATIONS 23

A.7. STATEMENTS 25

A.8. WHITE SPACE 28

A.9. NAMING CONVENTIONS 29

A.10. PROGRAMMING PRACTICES 30

A.11. CODE EXAMPLES 32

ANNEX B. QNX CODING STANDARDS FOR C / C++ 34

B.1. WHAT TO DO WHEN IT ISN'T SPECIFIED 34

B.2. FILE ORGANIZATION 34

B.3. SPECIFIC CODE FORMATTING ISSUES 38

B.4. COMMENTS 41

List of Figures

List of Tables

Table 1: Suggested components and their content in a user document 13

2010–2011 © Copyright lies with the respective authors and their institutions.

1 Introduction

1.1 Scope

This deliverable reports on the selection and usage of the software development infrastructure that will be used by all roles involved in the ARCOMEM project. The deliverable identifies the different needs in ARCOMEM software development in order to determine what tool infrastructure is required. The sole purpose of this choice is to enable the different development teams to meet their responsibilities and collaborate effectively with each other.

The factors considered in producing a list of recommended tools include tool usability and compatibility, ease of deployment and integration capability with other modules. Another important factor relates to possible feedback and experience on tool usage from partners in past or present RnD activities.

This deliverable also reports on the usage and adoption of these tools. We take extra steps in order to describe the procedures and processes for using infrastructure tools, providing guidelines and recommendations on how to collaborate within the ARCOMEM project. This includes guidelines on:

- Setting up the integrated development environment.
- Using the version control system.
- Building and releasing software.
- Using coding standards.
- Documenting software.

This guide should be used by all roles involved in ARCOMEM software development (e.g. developers, managers, technical authors etc.).

1.2 ARCOMEM Project Requirements

With the rapidly growing volume of resources in the Web, the dominating “collect-all” approach of Web archiving to preservation is an unsatisfying solution with respect to resources required and, especially, with respect to the usefulness of the archives. The goal of the project is to develop methods and tools for transforming digital archives into community memories based on novel, socially-aware preservation models. This will be done (a) by leveraging the Wisdom of the Crowds reflected in the Social Web context to drive concise and socially-aware content appraisal and selection processes taking events, entities and topics as seeds (b) and by using Social Web contextualization as well as extracted information on events, topics, perspectives, and entities in order to create richer and contextualized digital archives.

To achieve its goal, the ARCOMEM project will pursue the following scientific and technological objectives:

- Methods for Web mining and Social Web analysis, targeting the discovery of content appraisal within communities;
- Methods and tools for “by example” and descriptive archive content specification (based on topics, events and entities) and methods for the respective extension of crawling methods;

- Methods and tools for intelligent and adaptive decision support for content selection and content appraisal combining and inferring semantic knowledge from evidence extracted from archive content, the Social Web as well as archivists' feedback;
- Methods for enriching digital archive and especially Web archive content with consolidated information on covered events, involved entities and relevant topics, with information on opinions and perspectives as well as with contextualization information extracted from the Social Web;
- Methods for collaborative archive creation and extension based on mechanisms driving the Social Web;

The developed methods and formats will also be integrated with existing preservation technology and formats especially in the area of Web archiving (such as the WARC standard and the Heritrix Web Crawling engine). This includes the integration of the extended content appraisal and selection process with the process of crawling new archive content. In addition to the development of the methods mentioned above, the ARCOMEM team will create two socially-aware archiving applications for validating and showcasing ARCOMEM technology: one in the area of broadcasting archives, and one in the area of political content relevant for parliaments and archiving of political history.

From the previous discussion, it is obvious that the outcome of the project is a multimodal software system. Many of these modules will interact with each other, while others will be provided "as is" to the users. The two applications will be totally exposed to the users, requiring extra usability features. As the consortium comprises a large, diverse group of international partners, powerful collaboration platforms/tools as well as certain development rules are necessary to be established. The tools will enable distributed management of software development by the different groups, while ensuring a unified integration. Moreover, specific templates and usage rules/guidelines need to be set in order to minimize the integration cost, maximize performance and avoid redundant steps. It is the purpose of this document to identify the software development infrastructure used by ARCOMEM as well as relevant guidelines to be followed throughout the project's lifetime.

1.3 Document Structure

The rest of this document is structured as follows:

Section 2 reports on the selection of tools that comprise the ARCOMEM development infrastructure. Section 3 contains a collection of usage and best practice guidelines for the installed components as well as coding and documentation standards/recommendations. The document is summarized with the conclusions of Section 4.

2010–2011 © Copyright lies with the respective authors and their institutions.

2 ARCOMEM Software Development Infrastructure

The essential system required by the software development infrastructure comprises a software repository, based on a version control system. The ARCOMEM consortium consist of different partners that will develop potentially heterogeneous software modules. However, the software development process needs to be coordinated among the partners and managed by a central repository, in order to properly deliver the integrated versions of the technologies.

Given the previous practical experience of certain ARCOMEM partners, the ARCOMEM consortium decided to use the **Mercurial** system as source code repository for the ARCOMEM project. The choice we made within ARCOMEM is to use specific tools for different tasks in the development process, instead of using a single monolithic system covering the entire software development infrastructure. The automatic build and testing platform will be handled by the **Hudson** system, and we will use **Jira** as features/bugs tracking system.

2.1 Version Control System: Mercurial

Mercurial is a cross-platform, distributed revision control tool for software developers. It is mainly implemented using the Python programming language, but includes a binary diff implementation written in C. It is supported on Windows, and Unix-like systems including FreeBSD, OSX and Linux. Mercurial is primarily a command line program but graphical user interface extensions are available. All of Mercurial's operations are invoked as keyword options to its driver program “hg”, a reference to the chemical symbol of the element mercury.

As described in the public documentation of the system [1], as well as in its reference manual [2], the main features that recommend Mercurial can be summarized as following:

- Efficient for handling very large distributed projects.
- Decentralized system: using clone.
- Robust handling of both plain text and binary files.
- Free and open-source software.

From the point of view of ARCOMEM, these characteristics of the software repository fit well with the project's specificity and are in-line with the decentralized development process inside the consortium. We also chose Mercurial for its simplicity to use and its flexibility.

The quick start for using Mercurial contains the following 6 basic steps:
$ hg clone http://mercurial.internetmemory.org/projects/arcomem
$ cd arcomem
$ (edit files)
$ hg add (new files)
$ hg commit -m 'My changes'
$ hg push

2.2 Build and Integration Platform: Hudson

Hudson [3] is a continuous integration (CI) tool written in Java, which runs in a servlet container, such as Apache Tomcat or the GlassFish application server. It supports SCM tools including CVS, Subversion, Git and Mercurial and can execute Apache Ant and Apache Maven based projects, as well as arbitrary shell scripts and Windows batch commands. Hudson is free and open-source software, released under the MIT License. The Hudson project is supported by Oracle Corporation.

The tasks related to software integration and testing for the ARCOMEM project will be assigned to the Hudson system:

- Check-out the source code, build and test different modules of the project.
- Publish the results.
- Communicate the results to the team members.

2.3 Bug Tracking System: JIRA

JIRA [4] is a proprietary issue tracking product, developed by Atlassian. It is commonly used for bug tracking, issue tracking, and project management, to improve code quality and the speed of development. Atlassian provides JIRA for free to open source projects and organizations that are non-profit, non-government, non-academic, non-commercial, and non-political. For commercial customers, the full source code is available under a developer source license.

JIRA is written in Java and uses the webwork 1 technology stack. For Remote Procedure Calls (RPC), JIRA supports SOAP, XML RPC, and has a Java API. JIRA integrates with source control programs (SCM) such as Subversion, CVS, Clearcase, Visual SourceSafe, Mercurial, and Perforce.

2010–2011 © Copyright lies with the respective authors and their institutions.

3 ARCOMEM Software Development Best Practice Guide

3.1 Version Control Management

Repositories
A central server will be set up at the Internet Memory premises to hold the repositories. The access will be made over HTTP or HTTPS. To avoid constant merging or rebasing, each component will be in a repository of its own. All the repositories will be accessible with the same URL prefix (of the form https://some.server/hg) and a web interface at this URL will allow finding information about each repository (file browser, commit log, branches graph, etc). A separate repository will be created for general documentation.

The different teams will have to ask Internet Memory to perform the necessary administration actions every time a repository must be created or a user added. Then, they will be able to clone the repositories they need, and work locally, commit modifications, and push their modifications to the central server when they see fit, preferably on a regular basis to improve visibility.

Guidelines
A real user identifier must be used for every commit (no root@localhost, for instance). To do so, one may write the following in his ~/.hgrc:

```
[ui]
username = First_nameLast_name
```

The first line of the commit message should be a summary of the changes. (Only the first line is shown on the "log" page of the web interface or when running 'hg log'.)

As far as practical, each commit should be about just one modification (fix, feature addition...).

A new feature should be committed with its tests.

No commit should break the automated tests.

When several parts are developed concurrently in the same repository, using the rebase extension to avoid branching and merging is preferred.

It is not recommended to store files over a few MB in Mercurial. If large files are necessary to compile and run the tests for a component, an FTP server will be made available to upload them.

3.2 Automated Building and Testing

Dependencies
Some components will depend on others. It will most likely be inconvenient to enforce compatibility between them at all times (every time commits are pushed to the main server). To indicate which versions of the dependencies a component is known to work with, in each repository's root directory a text file named dependencies.txt should list the other components it depends on, along the with the version it has been tested with.

Example: if component x depends on component y version v3 and component z version v2, x/dependencies.txt will contain:
y@v3
z@v2

This gives some flexibility: using "tip", one can specify the latest version, with a hash, one can specify any specific version in any branch and possibly decide to go back to a previous one if a problem is discovered.

Build
All components should run on Debian Linux Lenny. For practical purposes, the project's components should be compatible with the default Lenny packages.

To streamline automatic testing, the compilation of each project should be kept simple. In particular, only one step should be necessary for the main operations, for instance, 'make' to build, 'make test' to run all tests, 'make dist' to build .so files, jars...

An execution of the tests must end with a 0 status code if and only if all of them passed.

When external systems are necessary, the test code should set up as many things as possible: creating databases tables, filling them with sample data, running simulators... and perform the necessary clean up after testing, whether the tests passed or not.

Anything that could fail should be included: compilation and test suite runs, but also documentation generation, for instance. Only a shell command to launch the step and exit with 0 if and only if the step succeeded is required.

For each component, the build steps in Hudson will include:
- Running a program that will pull the correct version of all its dependencies (as specified in dependencies.txt) and recurse, copy all auxiliary files from the FTP server directory for each component to the build directory (mirroring the directory hierarchy of what was uploaded to the FTP server),
- Compile each component,
- Running the test suite.

The sources, logs and build artifacts will be available for each build (successful or not) through Hudson's web interface. It suffices to include any necessary step to the build process to make a release available ('ant dist', for instance). A limited number of builds can be saved (the last N builds).

A build will be triggered by a change of the specific component in the main repository (not its dependencies). It will also be possible to manually request a build at any time. Build status can be reported by e-mail.

3.3 Features/Bug Tracking

A single project comprised of several components will be used. All users will have the right to create, modify, comment and assign issues in all components. Any modification to an issue will trigger an e-mail notification to the assignee.

Possible issues type will include: bug, improvement, new feature. The states will include: open, in progress, closed, reopened.

2010–2011 © Copyright lies with the respective authors and their institutions.

3.4 Coding Guidelines

Please refer to Annex A for a thorough discussion relative to coding guidelines and conventions.

3.5 Documentation

All large software development projects generate a large amount of associated documentation with a high proportion of the process costs being incurred in producing this documentation. Documentation in such large projects is of vital importance as errors or omissions may lead to faulty usage, system failures, increased development costs and functionality mismatches [5]. Therefore, managers and software engineers should pay as much attention to documentation and its associated costs as to the development of the software itself. This section provides some guidelines and rules of thumb in creating documentation for the purposes of the ARCOMEM project, with some relevant ideas and structure adopted from [6] and [7].

The project documentation can be categorized as belonging to either public or internal documentation, relative to the level of dissemination of the respective work/deliverable:

1. Public documentation: Documentation for public circulation (e.g., deliverables, user guides, admin guides, online help, web content, etc.).
2. Internal documentation: Documentation that is created and used by software development teams but not distributed publicly (for example, code documentation, API specifications, architectural drafts, etc.).

Relative to its content, we can also generally divide documentation produced into two classes:

1. Process documentation: These documents record the process of development and maintenance. Plans, schedules, process quality documents and organizational and project standards are process documentation.
2. Product documentation: This documentation describes the product that is being developed. System documentation describes the product from the point of view of the engineers developing and maintaining the system; user documentation provides a product description that is oriented towards system users.

In this deliverable, we will refer to guidelines referring to product or software documentation. Process documentation is produced so that the development of the system can be managed. Within ARCOMEM, there exists specific provisioning for such tasks through the respective WPs (i.e., WP12 and WP13) and their deliverables. Product documentation is used after the system is operational but is also essential during system development. The creation of a documentation document, such as system specification, may represent an important milestone in the software development process (e.g., Deliverable 7.3 of ARCOMEM).

3.5.1 Product (Software) Documentation

Product documentation is concerned with describing the delivered software product. It must evolve in step with the product which it describes. Product documentation includes user documentation which tells users how to use the software product and system documentation which is principally intended for maintenance engineers.

3.5.1.1 User Documentation

There exists a variety of users and levels of detail expertise that user documentation needs to be compiled upon. The most important categorization distinguishes users between end-users and administrators. To cover the full spectrum of users, a set of documents relative to the following areas must be compiled:

1. *The functional description* of the system outlines the system requirements and briefly describes the services provided.
2. The *system installation document* is intended for system administrators. It should provide details of how to install the system in a particular environment.
3. The *user manual* should present an informal introduction to the system, describing its 'normal' usage. It should describe how to get started and how end-users might make use of the common system facilities.
4. The *system reference manual* should describe the system facilities and their usage with a complete listing of error messages and recovery options.

3.5.1.2 System Documentation

System documentation includes all of the documents describing the system, such as documents describing the design, implementation and testing of a system or its components. For large systems, such as the one developed in the ARCOMEM project, the system documentation should include:

1. The requirements document and an associated rationale.
2. A document describing the system architecture. (These two are described in D1.1 and D1.2 of the project)
3. For each program in the system, a description of the architecture of that program.
4. For each component in the system, a description of its functionality and interfaces.
5. Program source code listings. These should be commented to explain complex sections of code and provide a rationale for the coding method used.
6. Validation documents describing how each program is validated and how the validation information relates to the requirements.
7. A system maintenance guide which describes known problems and how evolution of the system has been taken into account in its design.

3.5.1.3 Document Structure and Recommendations

Document structure has a major impact on readability and usability and it is important to design this carefully when creating documentation. Document structures should also be designed so that the different parts are as independent as possible. Structuring a document properly also allows readers to find information more easily.

The IEEE standard for user documentation [6] proposes that the structure of a document should include the components shown in Table 1. The standard makes clear that these are desirable or essential features of a document but the final structure and content depend on the designers of the documentation.

Table 1: Suggested components and their content in a user document

Component	Description
Identification data	Data such as a title and identifier that uniquely identifies the document.
Table of contents	Chapter/section names and page numbers.
List of illustrations	Figure numbers and titles
Introduction	Defines the purpose of the document and a brief summary of the contents
Information for use of the documentation	Suggestions for different readers on how to use the documentation effectively.

2010–2011 © Copyright lies with the respective authors and their institutions.

Concept of operations	An explanation of the conceptual background to the use of the software.
Procedures	Directions on how to use the software to complete the tasks that it is designed to support. on software
Information on the software commands	A description of each of the commands supported by the software.
Error messages and problem resolution	A description of the errors that can be reported and how to recover from these errors.
Glossary	Definitions of specialized terms used.
Related information sources	References or links to other documents that provide additional information
Navigational features	Features that allow readers to find their current location and move around the document.
Index	A list of key terms and the pages where these terms are referenced.
Search capability	In electronic documentation, a way of finding specific terms in the document.

Like all standards, this documentation standard has to be adapted to the situation where it is used. These should instantiate the advice in the standard to the local situation and define the specific structures and formats that should be used.

3.5.1.4 Technical Documentation Guidelines

Technical documentation explains the source code. When creating software, some textual, natural language information is necessary. As maintaining internal technical documentation is quite difficult, automation is essential. There exists a variety of tools that enable software developers to write this documentation as source code comments. The output format is usually in a reference guide style. Technical documentation should be created using the same tool as for the regular source code, so that the programmer can refer to their code directly.

There are different forms of technical documentation. This is due to the different purposes of the documentation (description of the source code, database tables, etc) and different tools used when documenting.

- Source code documentation should be written in parallel to the code itself, so it is important to create this documentation using the same tool to make this process as fast as possible. The output of the source code documentation should be generated automatically (with each release) and presented in an organized structure (with links and indexes), e.g. html format. The quality of the source code documentation depends on developer effort. Therefore, regular effort is crucial. The following source code elements should be documented:
  - Classes (author, date, aim of the class).
  - Fields.
  - Methods.
- Database documentation can be limited to a legible ER diagram (Entity-relationship diagram) and a description of tables, attributes and constraints.
- Description of the algorithms and data structures: All implemented algorithms and non-standard data structures should be documented in the source code and also in an external text file. For algorithms mechanisms and computational complexity (time and space) should be provided.
- Release notes: When software is developed in an iterative approach, each release should be published with basic information about new features, removed bugs, etc. An agile

approach assumes that release notes are generated automatically. Automation can be easily achieved when using advanced issue tracking system (e.g., JIRA).

3.5.2 Documentation Tools

Documentation can be created with tools that allow easy creation, modification and maintenance of a document's structure and content. In the following, we briefly describe tools that are recommended for documentation.

3.5.2.1 DocBook and DocuWiki

DocBook [8] provides a system for writing structured documents using SGML or XML. It is particularly well-suited to books and papers about computer hardware and software, though it is by no means limited to them. DocBook is a document type definition (DTD). Because it is a large and robust DTD, and because its main structures correspond to the general notion of what constitutes a document, DocBook has been adopted by a large and growing community of authors. DocBook files are used to generate the output in a wide number of formats (HTML, XSL-FO for later conversion into PDF), usually using DocBook XSL stylesheets. These stylesheets enable the generation of tables of contents, glossaries and indexes. DocBook is supported "out of the box" by a number of commercial tools and rapidly growing by a number of free software environments. In short, DocBook is an easy-to-understand and widely used DTD.

On the other hand, DocuWiki [9] is a standards-compliant, simple-to-use wiki which allows users to create rich documentation repositories. It provides an environment for individuals and teams and companies to create and collaborate using a simple yet powerful syntax that ensures data files remain structured and readable outside the wiki. Unlimited page revisions allow restoration to any earlier version. No database is required as data is stored in plain text files.

3.5.2.2 Javadoc, perldoc, Doxygen, ROBODoc, Sphinx

Javadoc [10], Doxygen [11], ROBODoc [12], etc are source code documentation tools that enable programmers to create technical documentation from source code comments. These tools are very similar and automatically generate hierarchical, hyperlinked documents. Developers should provide comments in a strictly defined format before classes, methods and fields that are to be documented. The resulting internal technical documentation can be rendered into several universal formats, e.g. XML, HTML and RTF.

3.5.2.3 Simple Text Documents

Natural documents (in text or other binary format, e.g., Word or Notepad) should be used when providing semi-formal requirements and documentation. This type of documentation can also be applied, when describing algorithms that have been implemented. It is a good practice to store the different versions of these files in the project repository, so conceptions and applied changes can be tracked.

2010–2011 © Copyright lies with the respective authors and their institutions.

4 Conclusions

This deliverable identifies the ARCOMEM software development infrastructure that will be used for the duration of the project in order to facilitate joint development at increased efficiency from varying work groups. Specifically, we describe the tools that will be used for version control management, the platform used for building and code integration and the bug tracking system. It also contains guidelines and recommendations on how to collaborate within ARCOMEM and use the aforementioned tools in the software development lifecycle.

5 References

1. Mercurial Website – http://mercurial.selenic.com
2. "Mercurial: The Definitive Guide", by Bryan O'Sullivan – http://hgbook.red-bean.com
3. http://hudson-ci.org/
4. http://www.atlassian.com/software/jira
5. Sommerville, I., Software Documentation, in Software Engineering, Vol. 2: The Supporting Processes. R. H. Thayer and M. I. Christensen (eds), Wiley-IEEE Press.[Book chapter], 2002.
6. IEEE, 2001. *Draft Standard for Software User Documentation*, IEEEStd1063/D5.1. 2001. New York: Institute of Electrical and Electronics Engineers.
7. GN3 Software Developer Best Practice Guide, Document code:GN3-09-186v2, http://www.geant.net/Media_Centre/Media_Library/Pages/Deliverables.aspx
8. DocBook: The source for Documentation: http://www.docbook.org/
9. dokuwiki: http://www.dokuwiki.org/dokuwiki
10. Javadoc Tool: http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
11. Doxygen: http://www.doxygen.org/
12. ROBODoc: Automating the software documentation process: http://www.xs4all.nl/~rfsber/Robo/robodoc.html

Annex A. Java Programming Coding Standards[1]

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

A.1. File Names

File Suffixes

Java Software uses the following file suffixes:

File Type	Suffix
Java source	`.java`
Java bytecode	`.class`

Common File Names

Frequently used file names include:

File Name	Use
`GNUmakefile`	The preferred name for makefiles. We use `gnumake` to build our software.
`README`	The preferred name for the file that summarizes the contents of a particular directory.

A.2. File Organization

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

Files longer than 2000 lines are cumbersome and should be avoided.

A.3. Java Source Files

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

Java source files have the following ordering:

- Beginning comments
- Package and Import statements

[1] http://java.sun.com/docs/codeconv/html/CodeConventions.doc.html#16711

2010–2011 © Copyright lies with the respective authors and their institutions.

- Class and interface declarations

Beginning Comments

All source files should begin with a c-style comment that lists the class name, version information, date, and copyright notice:

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

Package and Import Statements

The first non-comment line of most Java source files is a `package` statement. After that, `import` statements can follow. For example:

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

Note: The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.

Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear.

	Part of Class/Interface Declaration	**Notes**
1	Class/interface documentation comment (`/**...*/`)	
2	`class` or `interface` statement	
3	Class/interface implementation comment (`/*...*/`), if necessary	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment.
4	Class (`static`) variables	First the `public` class variables, then the `protected`, then package level (no access

		modifier), and then the `private`.
5	Instance variables	First `public`, then `protected`, then package level (no access modifier), and then `private`.
6	Constructors	
7	Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).

A.4. Lines

Line Length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

Note: Examples for use in documentation should have a shorter line length-generally no more than 80 characters.

Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking method calls:

```
someMethod(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);

var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                          longExpression3));
```

2010–2011 © Copyright lies with the respective authors and their institutions.

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
           + 4 * longname6; // PREFER

longName1 = longName2 * (longName3 + longName4
                        - longName5) + 4 * longname6; // AVOID
```

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}
```

Line wrapping for `if` statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
//DON'T USE THIS INDENTATION
if ((condition1 andand condition2)
    || (condition3 andand condition4)
    ||!(condition5 andand condition6)) { //BAD WRAPS
    doSomethingAboutIt();            //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 andand condition2)
        || (condition3 andand condition4)
        ||!(condition5 andand condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 andand condition2) || (condition3 andand condition4)
        ||!(condition5 andand condition6)) {
    doSomethingAboutIt();
}
```

Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
```

```
        : gamma;
```

A.5. Comments

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by /*...*/, and //. Documentation comments (known as "doc comments") are Java-only, and are delimited by /**...*/. Doc comments can be extracted to HTML files using the javadoc tool.

Implementation comments are meant for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective. to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or non-obvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters.

Comments should never include special characters such as form-feed and backspace.

Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

```
/*
 * Here is a block comment.
 */
```

Block comments can start with `/*-`, which is recognized by **indent**(1) as the beginning of a block comment that should not be reformatted. Example:

2010–2011 © Copyright lies with the respective authors and their institutions.

```
/*-
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 *    one
 *        two
 *            three
 */
```

Note: If you don't use **indent**(1), you don't have to use /*- in your code or make any other concessions to the possibility that someone else might run **indent**(1) on your code.

Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Java code:

```
if (condition) {

    /* Handle the condition. */
    ...
}
```

Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here's an example of a trailing comment in Java code:

```
if (a == 2) {
    return TRUE;            /* special case */
} else {
    return isPrime(a);      /* works only for odd a */
}
```

End-Of-Line Comments

The // comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow:

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else {
    return false;          // Explain why here.
```

```
}
//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

Documentation Comments

For further details, see "How to Write Doc Comments for Javadoc" which includes information on the doc comment tags (@return, @param, @see): at

http://java.sun.com/javadoc/writingdoccomments

For further details about doc comments and javadoc, see the javadoc home page at:

http://java.sun.com/javadoc/

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters `/**...*/`, with one comment per class, interface, or member. This comment should appear just before the declaration:

```
/**
 * The Example class provides ...
 */
public class Example { ...
```

Notice that top-level classes and interfaces are not indented, while their members are. The first line of doc comment (/**) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter.

If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment or single-line comment immediately *after* the declaration. For example, details about the implementation of a class should go in in such an implementation block comment *following* the class statement, not in the class doc comment.

Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with thefirst declaration *after* the comment.

A.6. Declarations

Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
int level; // indentation level
int size;  // size of table
```

is preferred over

2010–2011 © Copyright lies with the respective authors and their institutions.

```
int level, size;
```

Do not put different types on the same line. Example:

```
int foo,  fooarray[]; //WRONG!
```

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

```
int     level;          // indentation level
int     size;            // size of table
Object  currentEntry;   // currently selected table entry
```

Initialisation

Try to initialise local variables where they're declared. The only reason not to initialise a variable where it's declared is if the initial value depends on some computation occurring first.

Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void myMethod() {
    int int1 = 0;          // beginning of method block

    if (condition) {
        int int2 = 0;      // beginning of "if" block
        ...
    }
}
```

The one exception to the rule is indexes of `for` loops, which in Java can be declared in the `for` statement:

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;     // AVOID!
        ...
    }
    ...
```

```
}
```

Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

- No space between a method name and the parenthesis "(" starting its parameter list
- Open brace "{" appears at the end of the same line as the declaration statement
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}

    ...
}
```

- Methods are separated by a blank line

A.7. Statements

Simple Statements

Each line should contain at most one statement. Example:

```
argv++;       // Correct
argc--;       // Correct
argv++; argc--;       // AVOID!
```

Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "`{ statements }`". See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure, such as a `if-else` or `for` statement. This makes it easier to add

2010–2011 © Copyright lies with the respective authors and their institutions.

statements without accidentally introducing bugs due to forgetting to add braces.

return Statements

A `return` statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;

return myDisk.size();

return (size ? size : defaultSize);
```

if, if-else, if else-if else Statements

The `if-else` class of statements should have the following form:

```
if (condition) {
statements;
}

if (condition) {
statements;
} else {
statements;
}

if (condition) {
statements;
} else if (condition) {
statements;
} else {
statements;
}
```

Note: `if` statements always use braces {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
statement;
```

for Statements

A `for` statement should have the following form:

```
for (initialisation; condition; update) {
statements;
}
```

An empty `for` statement (one in which all the work is done in the initialisation, condition, and update clauses) should have the following form:

```
for (initialisation; condition; update);
```

When using the comma operator in the initialisation or update clause of a `for` statement, avoid the

complexity of using more than three variables. If needed, use separate statements before the `for` loop (for the initialisation clause) or at the end of the loop (for the update clause).

while Statements

A `while` statement should have the following form:

```
while (condition) {
statements;
}
```

An empty `while` statement should have the following form:

```
while (condition);
```

do-while Statements

A `do-while` statement should have the following form:

```
do {
statements;
} while (condition);
```

switch Statements

A `switch` statement should have the following form:

```
switch (condition) {
case ABC:
statements;
    /* falls through */

case DEF:
statements;
    break;

case XYZ:
statements;
    break;

default:
statements;
    break;
}
```

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /* falls through */ comment.

2010–2011 © Copyright lies with the respective authors and their institutions.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

try-catch Statements

A `try-catch` statement should have the following format:

```
try {
statements;
} catch (ExceptionClass e) {
statements;
}
```

A `try-catch` statement may also be followed by `finally`, which executes regardless of whether or not the `try` block has completed successfully.

```
try {
statements;
} catch (ExceptionClass e) {
statements;
} finally {
statements;
}
```

A.8. White Space

Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

- Between sections of a source file
- Between class and interface definitions

One blank line should always be used in the following circumstances:

- Between methods
- Between the local variables in a method and its first statement
- Before a block or single-line comment
- Between logical sections inside a method to improve readability

Blank Spaces

Blank spaces should be used in the following circumstances:

A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true) {
    ...
}
```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

- The expressions in a for statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

- Casts should be followed by a blank space. Examples:

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3))
                                + 1);
```

A.9. Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

Identifier Type	Rules for Naming	Examples
Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project,	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese

2010–2011 © Copyright lies with the respective authors and their institutions.

	machine, or login names.	
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	class Raster; class ImageSprite;
Interfaces	Interface names should be capitalized like class names.	interface RasterDelegate; interface Storing;

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	run();
		runFast();
		getBackground();

Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed.
	Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters.
		`int i;`
		`char c;`
		`float myWidth;`

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 999;
		static final int GET_THE_CPU = 1;

A.10. Programming Practices

Providing Access to Instance and Class Variables

Don't make any instance or class variable public without good reason. Often, instance variables don't need to be explicitly set or gotten-often that happens as a side effect of method calls.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a `struct` instead of a class (if Java supported `struct`), then it's appropriate to make the class's instance variables public.

Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead. For example:

```
classMethod();              //OK
AClass.classMethod();       //OK
anObject.classMethod();     //AVOID!
```

Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

```
if (c++ = d++) {         // AVOID! (Java disallows)
    ...
}
```

should be written as

```
if ((c++ = d++) != 0) {
    ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

```
d = (a = b + c) + r;         // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b andand c == d)      // AVOID!
if ((a == b) andand (c == d)) // RIGHT
```

Returning Values

Try to make the structure of your program match the intent. Example:

```
if (booleanExpression) {
    return true;
} else {
    return false;
```

2010–2011 © Copyright lies with the respective authors and their institutions.

```
}
```

should instead be written as

```
return booleanExpression;
```

Similarly,

```
if (condition) {
    return x;
}
return y;
```

should be written as

```
return (condition ? x : y);
```

Expressions before `?' in the Conditional Operator

If an expression containing a binary operator appears before the `?` in the ternary `?:` operator, it should be parenthesized. Example:

```
(x >= 0) ? x : -x;
```

Special Comments

Use `XXX` in a comment to flag something that is bogus but works. Use `FIXME` to flag something that is bogus and broken.

A.11. Code Examples

Java Source File Example

The following example shows how to format a Java source file containing a single public class. Interfaces are formatted similarly.

```
/*
 * @(#)Blah.java        1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */
```

```
package java.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Class description goes here.
 *
 * @version    1.82 18 Mar 1999
 * @author     Firstname Lastname
 */
public class Blah extends SomeClass {
    /* A class implementation comment can go here. */

    /** classVar1 documentation comment */
    public static int classVar1;

/**
* classVar2 documentation comment that happens to be
     * more than one line long
     */
private static Object classVar2;

    /** instanceVar1 documentation comment */
    public Object instanceVar1;

    /** instanceVar2 documentation comment */
    protected int instanceVar2;

    /** instanceVar3 documentation comment */
    private Object[] instanceVar3;

    /**
     * ...constructor Blah documentation comment...
     */
    public Blah() {
       // ...implementation goes here...
    }

    /**
     * ...method doSomething documentation comment...
     */
    public void doSomething() {
        // ...implementation goes here...
}

    /**
     * ...method doSomethingElse documentation comment...
     * @param someParam description
     */
    public void doSomethingElse(Object someParam) {
        // ...implementation goes here...
    }
}
```

2010–2011 © Copyright lies with the respective authors and their institutions.

Annex B. QNX Coding Standards for C / C++[2]

Consistency is at the heart of readable source code. Mixed coding style is harder to maintain than any particular "bad" coding style. When working on existing source code (GNU, Dinkum, QNX4, Photon etc), the particular format and style used by that source and its associated modules (headers, other source files) should be maintained regardless of this document. The intention is not to retro fit a new style onto all of QNX's existing source code base.

For new projects this Annex provides a common set of C coding standards and recommendations that should be followed. Standardizing will provide consistency, improve portability, reduce errors and help new developers/third parties and users reading the OKKAM source learn the "OKKAM way".

B.1. What To Do When It Isn't Specified

This section provides a number of guidelines based on existing industry best practices; it does not cover every single case imaginable. When something is not specified (ie continuation behaviour of long function names or long argument lists) and there are no examples in the existing source to follow (Rule #1 Be Consistent) make a choice based on providing a readable result that you will be able to use consistently.

B.2. File Organization

There is no maximum length limit for files, but files with more than about 1000 lines are cumbersome to deal with. Lines longer than 79 columns are not handled well by all terminals and should be avoided if possible. The maximum line length is 132 columns.

Source File Naming Conventions

Source file names are made up of a base name, a period and a suffix. File names must be lower-case, do not have spaces, contain only a single 'dot' and have meaningful names. Avoid using file or directory names that can cause problems for certain filesystems: CON, PRN, AUX, CLOCK$, NUL, COM1-COM9, LPT1-LPT9 should be avoided.

When working with DDK's the file names/directory organization should be adhered to.

Some compilers and tools require certain suffix conventions for names of files. The following suffixes are required:

- C header file names must end in .h
- C source file names must end in .c
- C++ source file names must end in .cpp
- C++ header file names must end in .hpp
- Java source file names must end with .java
- Assembler source file names must end in .S or .s

[2] http://community.qnx.com/sf/wiki/do/viewPage/projects.core_os/wiki/QNXCodingStandard

- Yacc source file names end in .y
- Lex source file names end in .l
- ... add other types here...

Source Files

The order of sections for a program file are as follows:

- License/Copyright Header (refer to section 7)
- Module Description (refer to section 4.8)
- #include Directives
- typedefs
- externs
- globals
- Source code ....

Order of Functions Within a File

Deliberately arrange the order in which the functions are defined in a file so as to make forward declarations unnecessary and your source easier to maintain and follow. Typically, you do this by placing the function that calls most other static functions at the bottom of the source file and placing the functions at the bottom of the caller/called hierarchy towards the top of the file.

Header Files

Header files should keep the number of other headers they include to the bare minimum, subject to other rules - see below.

System Headers (i.e. headers files in /usr/include):

The order of sections for are as follows:

```
License/Copyright Header (refer to section 7)
Module Description (refer to section 4.8)
#ifndef _HEADERFILE_H_INCLUDED
#define _HEADERFILE_H_INCLUDED
... stuff in file ...

#endif
```

For header files in sub directories use the following syntax:

```
#ifndef _DIR_HEADERFILE_H_INCLUDED
#include <DIR/HEADERFILE.h>
#endif
```

where:

2010–2011 © Copyright lies with the respective authors and their institutions.

```
DIR = directory relative to /usr/include where the file will exist
HEADERFILE = name of the file
```

The special case of files in /usr/include/sys will have DIR = "" resulting in names such as __HEADERFILE.

For the special case of header files in /usr/include/sys

```
#ifndef __HEADERFILE_H_INCLUDED
#include <sys/HEADERFILE.h>
#endif
```

For all type references in public header files other than those that are fundamental types, the file must include the appropriate definitions for those types. (If these definitions are in some other header files, thosefiles must be included enclosed by the appropriate #ifndef ... #endif blocks.)

Whenever possible, public header files must avoid polluting the user name space with extra symbol names. This means that when using types, if there's a version of the typename in the implementer's reserved name space (leading underscore's), you should use that version rather than the public typename. All of the reserved name versions are defined by including <sys/platform.h>, so by including that file, you hopefully will be able to avoid including other headers and introducing more symbol names than necessary. E.g. do:

```
#include <sys/platform.h>

typedef _Uint32t  my_type;
```

rather than:

```
#include <inttypes.h>

typedef uint32_t my_type;
```

In headers, function prototypes parameter names should all be declared so that they are prefixed by __ . This is just a safeguard against unexpected macro expansions for common names in parameters.

When defining constants to be used in flags, make sure the constants clearly indicate how large the target flag variable is. i.e. if the target flag storage is 32 bits, add appropriately enough leading zeroes to the constant.

Surround any typedef's or function prototype definitions in a public header file using:

```
__BEGIN_DECLS

/* typedefs and function prototypes go here */

__END_DECLS
```

The __BEGIN_DECLS and __END_DECLS macros are obtained by including <sys/platform.h>.

New header files should as often as possible avoid installing to /usr/include to avoid pollution of the /usr/include directory with a lot of random stuff. Preferably, if the header files belong to a specific module, then the header could install to /usr/include/modulename (as a directory, with the actual headers under it).

Existing module subdirectories:

ppc, mips, x86, arm, sh
: CPU specific headers

hw
: headers describing hardware stuff

photon
: Photon headers

Application Header Files

The order of sections are as follows:

License/Copyright Header (refer to section 7)

Module Description (refer to section 4)

```
#ifndef HEADERFILE_H_INCLUDED
#define HEADERFILE_H_INCLUDED

... stuff in file ...

#endif
```

Application header files should never use types defined in platform.h. They should instead use inttypes.h. E.g. do:

```
#include <inttypes.h>

typedef uint32_t my_type;
```

rather than:

```
#include <sys/platform.h>

typedef _Uint32t  my_type;
```

2010–2011 © Copyright lies with the respective authors and their institutions.

B.3. Specific Code Formatting Issues

The following points address specific code formatting issues. CONSISTANCY with existing source formatting and style takes precedence over these guidelines.

Running a formatting utility like indent over source code should be considered only as a last resort to restore sanity to source code that has been irreparably infected with multiple formats and styles. In this case it is suggested to use these guidelines for the re-formatting.

Indention

The standard indention is four (4) spaces.

If...Else Statements

The else clause of an if {...} else {...} should be "cuddled" as shown in this example:

```
if (something) {
        /* Do some stuff here. */
} else {
        /* Do something else. */
}
```

Case Statements

Case statements should be "cuddled" as shown in this example:

```
switch (number) {
case 0:
        ...
        break;
case 1:
        ...
        /* Fall through */
default:
        break;
}
```

Continuations from one case to another should be clearly identified with a comment as above.

For Statements

For statements should be "cuddled" as shown in this example:

```
for (cnt = 0; cnt < some_value; cnt++) {
        ... do stuff ...
}
```

While Statements

While statements should be "cuddled" as shown in this example:

```
while (some_condition) {
        ... do stuff ...
}

do {
        ... do stuff ...
} while (some_condition);
```

Format of Data Structures

Do not make assumptions about the layout or padding of a structure or the allocated size of a data structure. These depend on the compiler implementation and can vary significantly with the type of target CPU.

When defining a structure that will be used in message passing or will be written to external device (e.g a hard disk, serial line), the types of all the fields must be explicitly sized. E.g:

struct foo { int bar; }; is a no-no, while for a PUBLIC header (ie /usr/include)

```
struct foo {
        _Int32t bar;
};
```

or NON-PUBLIC header

```
struct foo {
        int32_t bar;
};
```

is correct. Note that even when a _Int32t style declaration is used the documentation should use the int32_t as the reference type.

On the opposite side, do NOT use an explicitly sized type unless you really need it - internal structure definitions can get along quite happily without them and you don't limit yourself in the future when we're running on a 64-bit CPU (and that's going to be sooner than you think). It is also a good idea to plan for expansion when defining public data structures. i.e.

```
struct foo {
        int     bar;
        int     rsvd[4];
};
```

Data Alignment

Align your data, especially structure elements, on their native boundaries. This means that shorts should start on any even address, longwords on addresses evenly divisible by four, and 64-bit values (quadwords) on addresses evenly divisible by eight.

2010–2011 © Copyright lies with the respective authors and their institutions.

```
struct foo {
        int16_t    type;
        int16_t    rsvd;       /* Explicit padding/reserved */
        int32_t    data;
};
```

Module Description

Each source/header file should contain a block comment immediately after the license header that gives a short description of what the module does and (if not clear) how to use it. Discussion of non-trivial design decisions and side-effects is also appropriate. For example:

```
/*
 * options.c
 *
 * This file contains code to parse command line options.
 */
```

Function Description

Each function should be preceded by a block comment. The comment should provides usefull information about:

- A synopsis of what the function does
- Appropriate or in appropriate uses for the function
- Side effects (state changes, acquired locks) of calling the function
- Any non-trivial design decisions or historical rationale for behaviour

The comment block should be formatted in a manner such that it is easy to read and automated documentation generation tools such as doxygen can extract meaningful developer information. Avoid dup-licating information clear from the code. One doxygen example format:

```
/**
 * function_name A short synopsis of the function
 * - Description: This is a longer description of what
 * the function does.
 * - Arguments:
 *   - param1: The first parameter description
 *   - param2: The second parameter description
 * - Returns:
 *   - Something that comes back
 */
```

Real world example:

```
/**
 * smc9000_parse_options: command line option processing
 * - Description: This function handles the special exception
```

```
* cases in the general option processing for the SMC driver.
* - Arguments:
*   - opt: a pointer to the command line options
* - Returns:
*   - Option parsing status: EOK on success, EINVAL on failure
*/
int smc9000_parse_options( char *opt )
```

Function Naming

Public functions should be named using all lower case letters and underscores.

B.4. Comments

C style (/* */) and C++ style (//) comments are allowed. The only exception is that C++ style comments are not permited in public header files.

Short/Single Line Comments

Very short comments may appear on the same line as the code they describe, and should be tabbed over to separate them from the statements. If more than one short comment appears in a block of code they should all be tabbed to the same tab setting.

```
if (a == EXCEPTION) {
        b = TRUE; /* special case */
} else {
        b = isprime(a); /* works only for odd a */
}
```

Formatting Block Comments

Format block comments as follows:

```
/*
 * This is a block comment. Don't embellish these with lots of "stars
 * and bars."
 */
```

Some Comments about Comments

Comments must be in English and should tell the reader something non-obvious. A comment that repeats what is blindingly obvious is annoying at best. The following is an example:

```
counter += 10;    /* Add ten to counter */
```

It is often better to aggregate comments about high-level and architectural issues in one place, to allow the reader and maintainers of your code to learn much more in a shorter time than if they had to piece together the issues and ideas about your features from comments strewn throughout several files.

2010–2011 © Copyright lies with the respective authors and their institutions.